**Top Spring Boot Interview Questions and Answers**

User full link: <https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/howto-embedded-web-servers.html>

**management.endpoints.web.exposure.include=\* (enable actuators for all endpoints)**

**spring.main.web-application-type=none (to make Spring-boot as normal java app)**

**Spring Boot Interview Questions**

**1. What is Spring Boot?**

Spring Boot is built on the top of Spring framework to create stand-alone RESTful web application with very minimal configuration and there is no need of external servers to run the application, because it has embedded servers like Tomcat and Jetty etc.

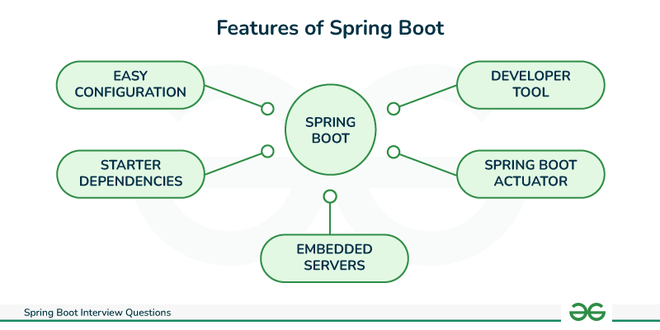
* Spring Boot framework is independent.
* It creates executable spring applications that are production-grade.

**To know more about Spring Boot, refer to this article** – [Introduction to Spring Boot](https://www.geeksforgeeks.org/introduction-to-spring-boot)

**2. What are the Features of Spring Boot?**

There are many useful features of Spring Boot. Some of them are mentioned below:

* **Auto-configuration –** Spring Boot automatically configures dependencies by using **@EnableAutoconfiguration** annotation and reduces boilerplate code.
* **Spring Boot Starter POM** – These Starter POMs are pre-configured dependencies for functions like database, security, maven configuration etc.
* **Spring Boot CLI (Command Line Interface)** – This command line tool is generally for managing dependencies, creating projects and running the applications.
* **Actuator –**Spring Boot Actuator provides health check, metrics and monitors the endpoints of the application. It also simplifies the troubleshooting management.
* **Embedded Servers –**Spring Boot contains embedded servers like Tomcat and Jetty for quick application run. No need of external servers.



**3. What are the advantages of using Spring Boot?**

Spring Boot is a framework that creates stand-alone, production grade Spring based applications. So, this framework has so many advantages.

* **Easy to use:**The majority of the boilerplate code required to create a Spring application is reduced by Spring Boot.
* **Rapid Development:**Spring Boot’s opinionated approach and auto-configuration enable developers to quickly develop apps without the need for time-consuming setup, cutting down on development time.
* **Scalable:**Spring Boot apps are intended to be scalable. This implies they may be simply scaled up or down to match your application’s needs.
* **Production-ready:** Metrics, health checks, and externalized configuration are just a few of the features that Spring Boot includes and are designed for use in production environments.

**4. Define the Key Components of Spring Boot.**

The key components of Spring Boot are listed below:

* Spring Boot starters
* Auto-configuration
* Spring Boot Actuator
* Spring Boot CLI
* Embedded Servers

**5. Why do we prefer Spring Boot over Spring?**

Here is a table that summarizes why we use Spring Boot over Spring framework.

| **Feature** | **Spring** | **Spring Boot** |
| --- | --- | --- |
| **Ease of use** | More complex | Easier |
| **Production readiness** | Less production-ready | More production-ready |
| **Scalability** | Less scalable | More scalable |
| **Speed** | Slower | Faster |
| **Customization** | Less Customizable | More Customizable |

**To know more, refer to the article –**[**Difference between Spring and Spring Boot**](https://www.geeksforgeeks.org/difference-between-spring-and-spring-boot)

**6. Explain the internal working of Spring Boot.**

**Here are the main steps involved in how Spring Boot works:**

* Start by creating a new Spring Boot project.
* Add the necessary dependencies to your project.
* Annotate the application with the appropriate annotations.
* Run the application.

**To know more about internal working of spring boot application, refer to this article –**[How Spring Boot Application works Internally?](https://www.geeksforgeeks.org/how-spring-boot-application-works-internally)

**7. What are the Spring Boot Starter Dependencies?**

Spring Boot provides many starter dependencies. Some of them which are used the most in the Spring Boot application are listed below:

* Data JPA starter
* Web starter
* Security starter
* Test Starter
* Thymeleaf starter

**8. How does a spring application get started?**

A Spring application gets started by calling the **main()** method with **@SpringBootApplication**annotation in the **SpringApplication** class. This method takes a SpringApplicationBuilder object as a parameter, which is used to configure the application.

* Once the SpringApplication object is created, the **run()** method is called.
* Once the application context is initialized, the run() method starts the application’s embedded web server.

**Example:**

Java

**import** **org.springframework.boot.SpringApplication**;

**import** **org.springframework.boot.autoconfigure.SpringBootApplication**;

@SpringBootApplication

**public** **class** **MyApplication**

{

**public** **static** void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

**9. What does the @SpringBootApplication annotation do internally?**

The **@SpringBootApplication** annotation combines three annotations. Those three annotations are: **@Configuration, @EnableAutoConfiguration,**and **@ComponentScan**.

* **@AutoConfiguration**: This annotation automatically configuring beans in the class path and automatically scans the dependencies according to the application need.
* **@ComponentScan**: This annotation scans the components (@Component, @Service, etc.) in the package of annotated class and its sub-packages.
* **@Configuration:**This annotation configures the beans and packages in the class path.

@SpringBootApplication automatically configures the application based on the dependencies added during project creation and bootstraps the application by using run() method inside the main class of an application.

*@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan*

**10. What is Spring Initializr?**

**Spring Initializer** is a tool that helps us to create skeleton of spring boot project or project structure by providing a maven or gradle file to build the application. It set up the framework from scratch.

**11. What are Spring Boot CLI and the most used CLI commands?**

**Spring Boot CLI** is a command-line tool that can be used to **create, run,**and**manage** Spring Boot applications. It is a powerful tool that can help us to get started with Spring Boot quickly and easily. It is built on top of the Groovy programming language.

Most used **CLI commands** are:

* -run
* -test
* -jar
* -war
* –init
* -help

**Spring Boot Interview Questions for Intermediate**

**12. What are the basic Spring Boot Annotations?**

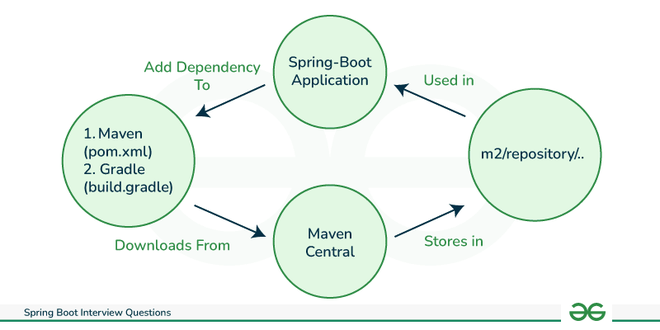
* **@SpringBootApplication:**This is the main annotation used to bootstrap a Spring Boot application. It combines three annotations: **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan**. It is typically placed on the main class of the application.
* **@Configuration:** This annotation is used to indicate that a class contains configuration methods for the application context. It is typically used in combination with @Bean annotations to define beans and their dependencies.
* **@Component:** This annotation is the most generic annotation for any Spring-managed component. It is used to mark a class as a Spring bean that will be managed by the Spring container.
* **@RestController:**This annotation is used to define a RESTful web service controller. It is a specialized version of the @Controller annotation that includes the @ResponseBody annotation by default.
* **@RequestMapping:**This annotation is used to map HTTP requests to a specific method in a controller. It can be applied at the class level to define a base URL for all methods in the class, or at the method level to specify a specific URL mapping.

**To know more about Spring Boot Annotations, refer to this article –**[Spring Boot – Annotations](https://www.geeksforgeeks.org/spring-boot-annotations)

**13. What is Spring Boot dependency management?**

**Spring Boot dependency management** makes it easier to manage dependencies in a Spring Boot project. It makes sure that all necessary dependencies are appropriate for the current Spring Boot version and are compatible with it.

*To create a web application, we can add the S****pring Boot starter web dependency****to our application.*



**To know more about Spring Boot Dependency Management, refer to this article –**[**Spring Boot – Dependency Management**](https://www.geeksforgeeks.org/spring-boot-dependency-management)

**14. Is it possible to change the port of the embedded Tomcat server in Spring Boot?**

Yes, it is possible to change the port of the embedded Tomcat server in a Spring Boot application.

The simple way is to set the **server. port** property in your application’s **application.properties** file. For example, to set the port to 8081, add the following property to the application.properties file:

server.port=8081

**15. What is the starter dependency of the Spring boot module?**

**Spring Boot Starters** are a collection of pre-configured maven dependencies that makes it easier to develop particular types of applications. These starters include,

* Dependencies
* Version control
* Configuration needed to make certain features.

To use a **Spring Boot starter dependency**, we simply need to add it to our project’s **pom.xml** file. For example, to add the Spring Boot starter web dependency, add the following dependency to the pom.xml file:

<dependency>  
 <groupId>org.springframework.boot</groupId>   
 <artifactId>spring-boot-starter-web</artifactId>   
</dependency>

**To know more about Spring Boot Starters, refer to this article –**[Spring Boot – Starters](https://www.geeksforgeeks.org/spring-boot-starters/?ref=)

**16. What is the default port of Tomcat in spring boot?**

The default port of the embedded Tomcat server in Spring Boot is **8080**. We can change the default port by setting the **server.port** property in your application’s **application.properties** file.

**17. Can we disable the default web server in the Spring Boot application?**

Yes, we can disable the default web server in the Spring Boot application. To do this, we need to set the **server.port** property to “-1” in the application’s **application.properties**file.

**18. How to disable a specific auto-configuration class?**

To disable a specific auto-configuration class in a Spring Boot application, we can use the **@EnableAutoConfiguration** annotation with the “**exclude”** attribute.

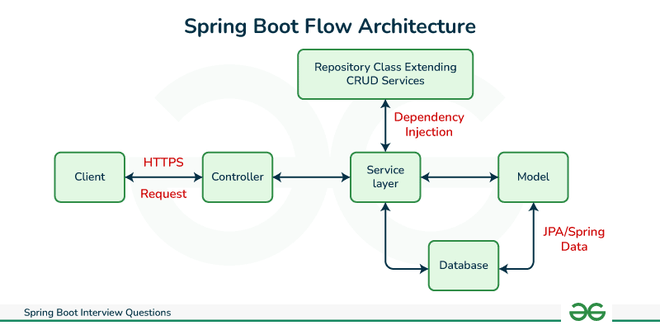
@EnableAutoConfiguration(exclude = {//classname})

**19. Can we create a non-web application in Spring Boot?**

Yes, we can create a non-web application in Spring Boot. Spring Boot is not just for web applications. Using Spring Boot, we can create applications like Microservices, Console applications, and batch applications.

**20. Describe the flow of HTTP requests through the Spring Boot application.**

The flow of HTTP requests through a Spring Boot application is as follows:



* First client makes an **HTTP request**(**GET, POST, PUT, DELETE**) to the browser.
* After that the request will go to the controller, where all the requests will be mapped and handled.
* After this in Service layer, all the **business logic** will be performed. It performs the business logic on the data that is mapped to **JPA (Java Persistence API)**using model classes.
* In repository layer, all the **CRUD** operations are being done for the **REST APIs**.
* A **JSP page** is returned to the end users if no errors are there.

**Actuals HTTPS FLOW Explanation**

* When explaining the flow of an HTTPS request from a client to a server in a Spring Boot application during an interview, you can break it down into the following steps:
* **1. Client Initiates HTTPS Request:**
* The client (e.g., a web browser or a REST client) initiates an HTTPS request to the server by specifying a URL that starts with https://.
* The request is sent to the server over the network, and it includes the desired resource (e.g., a REST endpoint) along with other details like HTTP method (GET, POST, etc.).
* **2. Server Receives HTTPS Request:**
* The server, running a Spring Boot application, is configured to listen for HTTPS requests on a specific port (commonly 443 for HTTPS or another specified port like 8443).
* The server has an SSL/TLS certificate installed, which is used to secure the connection.
* **3. TLS Handshake Process:**
* **Certificate Exchange:** The server responds to the client's request by sending its SSL certificate. This certificate contains the server's public key and is signed by a trusted Certificate Authority (CA).
* **Client Validation:** The client validates the server's certificate to ensure it is from a trusted source and has not been tampered with.
* **Session Key Generation:** If the certificate is valid, the client generates a session key (a symmetric key) and encrypts it using the server's public key. This encrypted session key is sent back to the server.
* **Server Decryption:** The server decrypts the session key using its private key. Now both the client and server have the same session key, which will be used for encrypting and decrypting data during this session.
* **4. Secure Data Transmission:**
* With the session key established, both the client and server use it to encrypt the data they exchange.
* The client sends the encrypted HTTP request (e.g., headers, parameters, body) to the server.
* The server decrypts the incoming request, processes it (e.g., invoking a controller in the Spring Boot application), and then generates a response.
* **5. Server Responds with Encrypted Data:**
* The server encrypts the HTTP response using the session key and sends it back to the client.
* The client receives the encrypted response and decrypts it using the session key.
* **6. Client Processes Response:**
* Once decrypted, the client can read the response data (e.g., HTML content, JSON data) and render it to the user or process it further.
* **7. Connection Closure:**
* After the data exchange is complete, the connection between the client and server may be closed, or it may be kept alive for further requests, depending on the connection settings (like keep-alive).
* **Key Points to Mention in an Interview:**
* **Security:** HTTPS ensures data confidentiality and integrity by encrypting the communication between the client and server.
* **TLS Handshake:** The handshake process is crucial for establishing a secure session, involving certificate exchange and session key generation.
* **Spring Boot Configuration:** Emphasize that Spring Boot simplifies setting up HTTPS with properties like server.ssl.key-store and server.ssl.key-password.
* **Performance Considerations:** While HTTPS provides security, it has overhead due to encryption and decryption, which should be considered in performance-sensitive applications.

**21. Explain @RestController annotation in Spring Boot.**

**@RestController** annotation is like a shortcut to building RESTful services. It combines two annotations:

* **@Controller**: Marks the class as a request handler in the Spring MVC framework.
* **@ResponseBody**: Tells Spring to convert method return values (objects, data) directly into HTTP responses instead of rendering views.

It enables us to Define endpoints for different **HTTP methods (GET, POST, PUT, DELETE),**return data in various formats (JSON, XML, etc.) and map the request parameters to method arguments.

**22. Difference between @Controller and @RestController**

| **Features** | **@Controller** | **@RestController** |
| --- | --- | --- |
| **Usage** | It marks a class as a controller class. | It combines two annotations i.e. @Controller and @ResponseBody. |
| **Application** | Used for Web applications. | Used for RESTful APIs. |
| **Request handling and Mapping** | Used with @RequestMapping annotation to map HTTP requests with methods. | Used to handle requests like GET, PUT, POST, and DELETE. |

*Note: Both annotations handle requests, but @RestController prioritizes data responses for building API.*

**23. What is the difference between RequestMapping and GetMapping?**

| **Features** | **@RequestMapping** | **@GetMapping** |
| --- | --- | --- |
| **Annotations** | @RequestMapping | @GetMapping |
| **Purpose** | Handles various types of HTTP requests (GET, POST, etc.) | Specifically handles HTTP GET requests. |
| **Example** | @RequestMapping(value = “/example”, method = RequestMethod.GET) | @GetMapping(“/example”) |

**24. What are the differences between @SpringBootApplication and @EnableAutoConfiguration annotation?**

| **Features** | **@SpringBootApplication** | **@EnableAutoConfiguration** |
| --- | --- | --- |
| **When to use** | When we want to use auto-configuration | When we want to customize auto-configuration |
| **Entry point** | Typically used on the main class of a Spring Boot application, serving as the entry point. | Can be used on any configuration class or in conjunction with @SpringBootApplication. |
| **Component Scanning** | Includes **@ComponentScan**annotation to enable component scanning. | Does not perform component scanning by itself. |
| **Example** | @SpringBootApplication public class MyApplication { public static void main(String[] args) { SpringApplication.run(MyApplication.class, args); } } | @Configuration @EnableAutoConfiguration public class MyConfiguration { } |

**25. What are Profiles in Spring?**

**Spring Profiles** are like different scenarios for the application depending on the environment.

* You define sets of configurations (like database URLs) for different situations (development, testing, production).
* Use the **@Profile** annotation to clarify which config belongs to where.
* Activate profiles with **environment variables** or **command-line** options.

To use Spring Profiles, we simply need to define the **spring.profiles.active** property to specify which profile we want to use.

**26. Mention the differences between WAR and embedded containers.**

| **Feature** | **WAR** | **Embedded containers** |
| --- | --- | --- |
| **Packaging** | Contains all of the files needed to deploy a web application to a web server. | It is a web application server included in the same JAR file as the application code. |
| **Configuration** | Requires external configuration files (e.g., web.xml, context.xml) to define the web application. | Uses configuration properties or annotations within the application code. |
| **Security** | Can be deployed to a web server that is configured with security features. | Can be made more secure by using security features that are provided by JRE. |

**Advanced Spring Boot Interview Questions**

**27. What is Spring Boot Actuator?**

**Spring Boot Actuator** is a component of the Spring Boot framework that provides production-ready operational monitoring and management capabilities. We can manage and monitor your Spring Boot application while it is running.

*Note: To use Spring Boot Actuator, we simply need to add the****spring-boot-starter-actuator****dependency to our project.*

**To know more about Actuator, refer to this article –**[Spring Boot Actuator](https://www.geeksforgeeks.org/spring-boot-actuator/?ref=)

**28. How to enable Actuator in the Spring boot application?**

Below are the steps to enable actuator in Spring Boot Application:

* Add Actuator dependency.
* Enable endpoints in application.properties.
* Run your Spring Boot app.

Now we can access Actuator endpoints at URLs on the management port.

**29. What is the purpose of using @ComponentScan in the class files?**

**@ComponentScan** annotation is used to tell Spring to scan a package and automatically detect Spring components, configurations, and services to configure.The @ComponentScan annotation can be used in the following ways:

* **Without arguments**
* **With basePackageClasses**
* **With basePackages**

**To know more about @ComponentScan annotation, refer to this article –**[Spring @ComponentScan Annotation with Example](https://www.geeksforgeeks.org/spring-componentscan-annotation-with-example/?ref=)

**30. What are the @RequestMapping and @RestController annotations in Spring Boot used for?**

**@RequestMapping:** @RequestMapping is used to map HTTP requests to handler methods in your controller classes. It can be used at the class level and method level. It supports mapping by:

* HTTP method – GET, POST, PUT, DELETE
* URL path
* URL parameters
* Request headers

**@RestController:**@RestController is a convenience annotation that combines **@Controller** and **@ResponseBody**. It indicates a controller where every method returns a domain object instead of a view.

*@RestController = @Controller + @ResponseBody*

**31. How to get the list of all the beans in your Spring boot application?**

* Using the **ApplicationContext**object in Spring Boot, we can retrieve a list of all the beans in our application.
* The ApplicationContext is responsible for managing the beans and their dependencies.

**32. Can we check the environment properties in your Spring boot application explain how?**

Yes, we can check the environment properties in our Spring Boot Application. The Environment object in a Spring Boot application can be used to check the environment’s properties.

Configuration settings for the application, includes:

* property files
* command-line arguments
* environment variables

We can get the Environment instance by calling the **getEnvironment()** method.

**33. How to enable debugging log in the spring boot application?**

To enable debugging log in Spring Boot Application, follow the below steps:

* **Add the logging level property to application.properties.**
* **Configure the log pattern to include useful information.**
* **Run the Spring Boot application.**

Using the actuator endpoint, the log level can also be changed at runtime.

Curl -X POST \http://localhost:8080/actuator/loggers/<logger-name>   
\ -H 'content-type: application/json' \-d '{"configuredLevel": "DEBUG"}'

**34. What is dependency Injection and its types?**

**Dependency Injection** (DI) is a design pattern that enables us to produce loosely coupled components. In DI, an object’s ability to complete a task depends on another object. There three types of dependency Injections.

* **Constructor injection:** This is the most common type of DI in Spring Boot. In constructor injection, the dependency object is injected into the dependent object’s constructor.
* **Setter injection:** In setter injection, the dependency object is injected into the dependent object’s setter method.
* **Field injection**: In field injection, the dependency object is injected into the dependent object’s field.

**To know more about Dependency Injection, refer to the article –**[Spring Dependency Injection with Example – GeeksforGeeks](https://www.geeksforgeeks.org/spring-dependency-injection-with-example/?ref=)

**35. What is an IOC container?**

An **IoC (Inversion of Control)** Container in Spring Boot is essentially a central manager for the application objects that controls the creation, configuration, and management of dependency injection of objects (often referred to as beans), also referred to as a DI (Dependency Injection) container.

**To know more about IOC Container, refer to the article –**[Spring – IoC Container](https://www.geeksforgeeks.org/spring-ioc-container/?ref=)

**36. What is the difference between Constructor and Setter Injection?**

| **Features** | **Constructor Injection** | **Setter Injection** |
| --- | --- | --- |
| **Dependency** | Dependencies are provided through constructor parameters. | Dependencies are set through setter methods after object creation. |
| **Immutability** | Promotes immutability as dependencies are set at creation. | Dependencies can be changed dynamically after object creation. |
| **Dependency Overriding** | Harder to override dependencies with different implementations. | Allows easier overriding of dependencies using different setter values. |

**Bonus Spring Boot Interview Questions and Answers**

**1. What is Thymeleaf?**

**Thymeleaf** is a Java-based server-side **template engine** used in Java web applications to render dynamic web pages. It is a popular choice for server-side templating in the Spring ecosystem, including Spring Boot.

**To know more about Thymeleaf, refer to this article –**[Spring Boot – Thymeleaf with Example](https://www.geeksforgeeks.org/spring-boot-thymeleaf-with-example/?ref=)

**2. Explain Spring Data and What is Data JPA?**

**Spring Data** is a powerful framework that can be used to develop data-oriented applications. It aims to simplify the development of data-centric applications by offering abstractions, utilities, and integration with various data sources.

* **Spring Data JPA:** This project provides support for accessing data from relational databases using JPA.

**3. Explain Spring MVC**

**MVC**stands for**Model, View,**and**Controller. Spring MVC**is a web MVC framework built on top of the Spring Framework. It provides a comprehensive programming model for building web applications.

**4. What is Spring Bean?**

An object that is managed by the Spring IoC container is referred to as a spring bean. A Spring bean can be any Java object.

**5. What are Inner Beans in Spring?**

An **Inner Bean**refers to a bean that is defined within the scope of another bean’s definition. It is a way to declare a bean inside the configuration of another bean, without explicitly giving it a unique identifier.

To define an Inner Bean in Spring, we can declare it as a nested <bean> element within the configuration of the enclosing bean.

**6. What is Bean Wiring?**

**Bean wiring**is a mechanism in Spring that is used to manage the dependencies between beans. It allows Spring to inject collaborating beans into each other. There are two types of Bean Wiring:

* Autowiring
* Manual wiring

**To know more about Autowiring, refer to the article –**[Spring – Autowiring](https://www.geeksforgeeks.org/spring-autowiring/?ref=)

**7. What Are Spring Boot DevTools Used For?**

**Spring Boot DevTools**provides a number of development-time features and enhancements to increase developers’ productivity and can be used for the following purposes:

* Automatic application restart
* Fast application startup:
* Actuator endpoints
* Additional development utilities

**To know more about Spring Boot DevTools, refer to the article –**[Spring Boot – DevTools](https://www.geeksforgeeks.org/spring-boot-devtools/?ref=)

**8. What error do you see if H2 is not present in the class path?**

Below is the error we see if H2 is not present in the class path:

java.lang.ClassNotFoundException: org.h2.Driver

**9. Mention the steps to connect the Spring Boot application to a database using JDBC.**

To connect an external database like MySQL or Oracle to a Spring Boot application using JDBC, we need to follow below steps:

* Add the dependency for the JDBC driver of the database.
* Create an application.properties file.
* Configure the database connection properties.
* Create a JdbcTemplate bean.
* Use the JdbcTemplate bean to execute SQL queries and statements.

**To know more, refer to this article –**[Spring Boot – CRUD Operations using MySQL Database](https://www.geeksforgeeks.org/spring-boot-crud-operations-using-mysql-database/?ref=)

**10. Mention the advantages of the YAML file over than Properties file and the different ways to load the YAML file in Spring boot.**

Advantages of YAML file over Properties file:

* Easy to edit and modify.
* Conciseness
* Supports Complex data types.

Different ways to load YAML file in Spring Boot:

* Using the @ConfigurationProperties annotation
* Using the YamlPropertiesFactoryBean class

**11. What Do you understand about Spring Data Rest?**

**Spring Data REST** is a framework that exposes Spring Data repositories as RESTful web services. It allows us to expose repositories as REST endpoints with minimal configuration by following Spring Data REST Technologies like **Spring Data** and **Spring MVC**.

**To know more about Spring Data REST, Please Refer to this article-**[Spring – REST Controller](https://www.geeksforgeeks.org/spring-rest-controller/?ref=)

**12. Why is Spring Data REST not recommended in real-world applications?**

Here are the reasons why not to choose Spring Data REST:

* **Performance** – Performance may not be optimal for very large-scale applications.
* **Versioning** – It can be difficult to version the REST APIs exposed by Spring Data REST.
* **Relationships** – Handling relationships between entities can be tricky with Spring Data REST.
* **Filtering** – There are limited options for filtering the results returned by the endpoints.

**13. How is Hibernate chosen as the default implementation for JPA without any configuration?**

Spring Boot automatically configures **Hibernate** as the default JPA implementation when we add the **spring-boot-starter-data-jpa** dependency to our project. This dependency includes the Hibernate JAR file as well as the Spring Boot auto-configuration for JPA.

**To know more about Hibernate and JPA, Refer to below articles:**

* [Hibernate Architecture](https://www.geeksforgeeks.org/hibernate-architecture/?ref=)
* [Java – JPA vs Hibernate](https://www.geeksforgeeks.org/java-jpa-vs-hibernate/?ref=)

**14. Explain how to deploy to a different server with Spring Boot?**

Below are the steps on how to deploy to a different server with Spring Boot:

* Step 1: **Build your Spring Boot application.**
* Step 2: **Create a deployment package.**
* Step 3: **Deploy the deployment package to the server.**
* Step 4: **Start the server.**

**Spring Boot Interview Questions – FAQs**

**Q. What will be the Spring Boot Interview Questions for 5 Years Experience?**

*In the interview, candidates with over 5 years of experience are primarily questioned about these concepts.*

1. ***Auto-configuration:****Spring Boot automatically configures beans based on project dependencies, saving time in setup.*
2. ***Starters****: Dependency management artifacts for easy integration of common Spring Boot features like web apps, data access, and security.*
3. ***Production-ready applications:****Spring Boot provides embedded servers, actuators, and metrics for creating production-ready apps.*
4. ***Best practices:****Use dependency injection, version control, and thorough testing when developing Spring Boot apps.*
5. ***Challenges****: Understanding auto-configuration and selecting appropriate dependencies may be challenging.*
6. ***Improving skills****: Enhance Spring Boot skills through documentation, conferences, and contributing to the project.*

**Q. What are the most common Spring Boot interview questions?**

*The most common Spring Boot interview questions are:*

* *What is Spring Boot?*
* *What are the advantages of using Spring Boot?*
* *What are the features of Spring Boot?*
* *How to create a Spring Boot application?*
* *What is the difference between Spring Boot and Spring Framework?*
* *What are the starter dependencies in Spring Boot?*
* *What is the purpose of the @SpringBootApplication annotation?*
* *What is the purpose of the @Configuration annotation?*
* *What is the purpose of the @Bean annotation?*
* *What is the purpose of the @Autowired annotation?*
* *What is the purpose of the @Value annotation?*
* *What is the purpose of the @Profile annotation?*
* *What is the purpose of the @EnableAutoConfiguration annotation?*
* *What is the default port of the embedded Tomcat server in Spring Boot?*
* *How to change the port of the embedded Tomcat server in Spring Boot?*
* *How to enable actuator in Spring Boot?*
* *How to access actuator endpoints in Spring Boot?*

**Q. How can I prepare for Spring Boot interview questions?**

*There are a few things you can do to prepare for Spring Boot interview questions:*

* *Learn about Spring Boot*
* *Practice answering common Spring Boot interview questions*
* *Create a Spring Boot project and experiment with the different features*
* *Attend Spring Boot meetups and conferences*
* *Join the Spring Boot community on Stack Overflow and other forums*

**Q. What will be the Spring Boot Interview Questions for 2 Years Experience?**

*For candidates with up to 2 years of experience, interviews will typically focus on the core concepts of Spring Boot, such as auto-configuration, starters, actuator, and CLI. Questions may also be asked about how to create, configure, run, and deploy Spring Boot applications.*

* *Basics of Spring Boot*
* *Components of Spring Boot*
* *Create a Spring Boot application*
* *Configure Spring Boot application*
* *Run a Spring Boot application*
* *Deploy a Spring Boot application*
* *Best practices for developing Spring Boot applications*

**Q. What will be the Spring Boot Interview Questions for 3 Years Experience?**

*In the interview, candidates with over 3 years of experience are primarily questioned about these concepts.*

* *What are the different ways to start a Spring Boot application?*
* *What are the different ways to configure Spring Boot applications?*
* *How to use Spring Boot starters?*
* *How to use Spring Boot actuator?*
* *How to use Spring Boot CLI?*
* *How to use Spring Boot in a microservices architecture?*
* *How to secure a Spring Boot application?*
* *How to troubleshoot Spring Boot applications?*

**Spring Annotations**

1. **@Component**
   * Marks a class as a Spring component, allowing Spring to automatically detect it through classpath scanning and register it as a bean in the Spring application context.
2. **@Service**
   * Specialization of @Component that marks a class as a service. It is used to indicate that the class holds business logic.
3. **@Repository**
   * Specialization of @Component that marks a class as a repository, which is a Data Access Object (DAO) that encapsulates storage, retrieval, and search behavior.
4. **@Controller**
   * Specialization of @Component that marks a class as a Spring MVC controller, capable of handling web requests.
5. **@RestController**
   * A combination of @Controller and @ResponseBody. It marks a class as a controller where every method returns a domain object instead of a view. Suitable for RESTful services.
6. **@Autowired**
   * Allows Spring to resolve and inject collaborating beans into a bean.
7. **@Qualifier**
   * Used in conjunction with @Autowired to specify which bean should be injected when multiple beans of the same type exist.
8. **@Value**
   * Injects values from properties files, system properties, or environment variables into fields.
9. **@Configuration**
   * Marks a class as a source of bean definitions. Methods annotated with @Bean inside this class will return beans that Spring manages.
10. **@Bean**
    * Indicates that a method produces a bean to be managed by the Spring container. Used within classes annotated with @Configuration.
11. **@Scope**
    * Specifies the scope of a bean, such as singleton, prototype, request, or session.
12. **@Primary**
    * Indicates that a bean should be given preference when multiple candidates are qualified to be autowired.
13. **@Lazy**
    * Marks a bean to be initialized lazily, meaning it will be created and initialized only when it is first requested.
14. **@DependsOn**
    * Indicates that a bean depends on the initialization of one or more other beans.
15. **@PostConstruct**
    * A lifecycle callback method that will be invoked after dependency injection is done.
16. **@PreDestroy**
    * A lifecycle callback method that will be invoked before the bean is removed from the context.

**Spring MVC Annotations**

1. **@RequestMapping**
   * Maps HTTP requests to handler methods in controllers. Can be applied at the class or method level.
2. **@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping**
   * Shortcut annotations for @RequestMapping with specific HTTP methods.
3. **@RequestParam**
   * Binds a web request parameter to a method parameter in a controller.
4. **@PathVariable**
   * Binds a URI template variable to a method parameter.
5. **@RequestBody**
   * Indicates that a method parameter should be bound to the body of the web request.
6. **@ResponseBody**
   * Indicates that the return value of a method should be used as the response body of the web request.
7. **@ModelAttribute**
   * Binds a method parameter or method return value to a named model attribute and then exposes it to a web view.
8. **@ResponseStatus**
   * Marks a method or exception class with a status code and reason that should be returned.

**Spring Boot Annotations**

1. **@SpringBootApplication**
   * A combination of @Configuration, @EnableAutoConfiguration, and @ComponentScan. It marks the main class of a Spring Boot application and triggers the auto-configuration.
2. **@EnableAutoConfiguration**
   * Enables Spring Boot’s auto-configuration mechanism, which attempts to automatically configure your Spring application based on the jar dependencies you have added.
3. **@ComponentScan**
   * Configures the packages to be scanned for Spring components, configurations, and services.
4. **@ConfigurationProperties**
   * Binds the properties defined in application.properties or application.yml files to a POJO class.
5. **@EnableScheduling**
   * Enables Spring’s scheduled task execution capability, allowing methods to be scheduled for execution at regular intervals.
6. **@EnableAsync**
   * Enables Spring’s asynchronous method execution capability, allowing methods to run in the background.
7. **@SpringBootTest**
   * Used in testing; it loads the complete application context for integration tests.
8. **@TestConfiguration**
   * Used to define beans and configurations specifically for test cases.
9. **@ConditionalOnProperty**
   * Configures a bean to be created only if a specified property has a specific value.
10. **@ConditionalOnMissingBean**
    * Configures a bean to be created only if a specified bean is not already present in the context.
11. **@RestControllerAdvice**
    * Combination of @ControllerAdvice and @ResponseBody, used to handle exceptions in RESTful web services.

**Spring Security Annotations**

1. **@EnableWebSecurity**
   * Enables Spring Security’s web security support and provides the Spring MVC integration.
2. **@Secured**
   * Specifies a list of roles that can access a method or class.
3. **@PreAuthorize**
   * Allows method access based on complex conditions using SpEL (Spring Expression Language).
4. **@PostAuthorize**
   * Similar to @PreAuthorize, but checks authorization after the method has been executed.
5. **@RolesAllowed**
   * Specifies the roles allowed to execute the annotated method.

**Spring Boot Starter Parent**

* **Configuration**: It allows us to maintain consistency of Java Version and other related properties.
* **Dependency Management:** It controls the versions of dependencies to avoid conflict.
* **Source encoding**
* **Default Java Version**
* **Resource filtering**
* **It also controls the default plugin configuration.**

The *spring-boot-starter-parent* project is a special starter project that provides default configurations for our application and a complete dependency tree to quickly build our *Spring Boot* project. It also provides default configurations for Maven plugins, such as *maven-failsafe-plugin*, *maven-jar-plugin*, *maven-surefire-plugin*, and *maven-war-plugin*.

Beyond that, it also inherits dependency management from *spring-boot-dependencies,*which is the parent to the s*pring-boot-starter-parent*.

**Managing Dependencies**

Once we’ve declared the starter parent in our project, we can pull any dependency from the parent by just declaring it in our *dependencies* tag. We also don’t need to define versions of the dependencies; Maven will download jar files based on the version defined for the starter parent in the parent tag.

**Give Some Starters which we use mostly**

**1. spring-boot-starter-web**

* **Why & When to Use:**
  + Use this starter when building web applications, including RESTful web services and MVC-based applications.
* **What It Contains:**
  + This starter includes:
    - Spring MVC (spring-web, spring-webmvc): Provides support for web applications, including REST endpoints and the MVC architecture.
    - Embedded web server (Tomcat by default): Allows the application to run as a standalone web application without the need for an external server.
    - Jackson: Provides support for JSON processing, making it easier to work with REST APIs.
    - Validation API: Enables bean validation.

**2. spring-boot-starter-data-jpa**

* **Why & When to Use:**
  + Use this starter when you want to interact with relational databases using JPA (Java Persistence API) and Hibernate.
* **What It Contains:**
  + This starter includes:
    - Spring Data JPA: Provides repository support for JPA.
    - Hibernate: The default JPA implementation used by Spring Boot.
    - Database Connection Pool (HikariCP): For managing database connections.
    - Spring ORM: Facilitates integration with JPA and Hibernate.
    - Spring JDBC: Provides JDBC support, which is often required even in JPA-based applications.

**3. spring-boot-starter-security**

* **Why & When to Use:**
  + Use this starter to secure your Spring Boot application, whether you are securing web endpoints, adding authentication and authorization, or integrating with OAuth2 and other security frameworks.
* **What It Contains:**
  + This starter includes:
    - Spring Security: Provides security features such as authentication, authorization, and protection against common attacks (CSRF, XSS).
    - Additional security configurations: Integrated easily with other parts of the Spring ecosystem.

**4. spring-boot-starter-thymeleaf**

* **Why & When to Use:**
  + Use this starter when building server-side rendered web applications with Thymeleaf as the templating engine.
* **What It Contains:**
  + This starter includes:
    - Thymeleaf: A modern server-side Java template engine for web and standalone environments.
    - Spring MVC: Integrated with Thymeleaf for rendering views.
    - Additional Thymeleaf-related dependencies: For working with Thymeleaf dialects and extensions.

**5. spring-boot-starter-test**

* **Why & When to Use:**
  + Use this starter to set up a comprehensive testing environment for your Spring Boot application, including unit tests, integration tests, and more.
* **What It Contains:**
  + This starter includes:
    - JUnit 5: The most popular testing framework for Java.
    - Spring Test & Spring Boot Test: Provides support for integration testing, loading the Spring context.
    - Mockito: A mocking framework for unit tests.
    - AssertJ: A fluent assertion library for writing more readable test cases.
    - Hamcrest: A matcher library for writing expressive unit tests.
    - JSONassert: For asserting JSON data in tests.
    - Testcontainers: For running integration tests with real databases or other services in Docker containers.

**6. spring-boot-starter-actuator**

* **Why & When to Use:**
  + Use this starter to add production-ready features to your Spring Boot application, such as monitoring and management over HTTP or JMX.
* **What It Contains:**
  + This starter includes:
    - Spring Boot Actuator: Provides endpoints to monitor and manage the application, including health checks, metrics, environment info, and more.
    - Micrometer: A metrics collection and export library that integrates with monitoring systems like Prometheus, Grafana, and others.

**7. spring-boot-starter-logging**

* **Why & When to Use:**
  + This is included by default in all Spring Boot applications for logging purposes.
* **What It Contains:**
  + This starter includes:
    - Logback: The default logging framework used by Spring Boot.
    - SLF4J: A facade for different logging frameworks, allowing you to change the underlying logging implementation without changing your application code.

**8. spring-boot-starter-amqp**

* **Why & When to Use:**
  + Use this starter when you want to build applications that communicate with RabbitMQ using the AMQP protocol.
* **What It Contains:**
  + This starter includes:
    - Spring AMQP: Provides support for building messaging applications with RabbitMQ.
    - Spring Rabbit: A template for sending and receiving messages via RabbitMQ.

**9. spring-boot-starter-validation**

* **Why & When to Use:**
  + Use this starter when you need to validate user input, such as form data or API requests.
* **What It Contains:**
  + This starter includes:
    - Hibernate Validator: The reference implementation of the Bean Validation API.
    - Bean Validation API: Allows you to use annotations like @NotNull, @Size, @Email, etc., to validate data.

**10. spring-boot-starter-data-mongodb**

* **Why & When to Use:**
  + Use this starter when working with MongoDB as your NoSQL database.
* **What It Contains:**
  + This starter includes:
    - Spring Data MongoDB: Provides repository support for MongoDB.
    - MongoDB Driver: The Java driver for connecting and interacting with MongoDB databases.

These starters provide a convenient way to set up and configure your Spring Boot application with the necessary dependencies and are designed to cover common development scenarios. By including these starters, you can focus more on your application's business logic and less on configuration and dependency management.

**Embedded Web Servers**

Each Spring Boot web application includes an embedded web server. This feature leads to a number of how-to questions, including how to change the embedded server and how to configure the embedded server. This section answers those questions.

**76.1 Use Another Web Server**

Many Spring Boot starters include default embedded containers.

* For servlet stack applications, the spring-boot-starter-web includes Tomcat by including spring-boot-starter-tomcat, but you can use spring-boot-starter-jetty or spring-boot-starter-undertow instead.
* For reactive stack applications, the spring-boot-starter-webflux includes Reactor Netty by including spring-boot-starter-reactor-netty, but you can use spring-boot-starter-tomcat, spring-boot-starter-jetty, or spring-boot-starter-undertow instead.

When switching to a different HTTP server, you need to exclude the default dependencies in addition to including the one you need. Spring Boot provides separate starters for HTTP servers to help make this process as easy as possible.

Spring Boot starters are a set of convenient dependency descriptors that you can include in your application. They bring in a group of related dependencies and configurations, which simplifies the setup of common development tasks. Here are some of the most commonly used Spring Boot starters, along with explanations of why and when they are used, and what they contain:

**Q) How can make a spring boot Application as Normal Java Application**

A) In Property file add below property, remove web starter or any other dependency

**spring.main.web-application-type=none**

**Q) Use a Random Unassigned HTTP Port**

**To scan for a free port (using OS natives to prevent clashes) use server.port=0.**

**Q) Disabling the Embedded Server:**

* **Setting server.port=-1 instructs Spring Boot to disable the embedded web server. This means that the application will not start any server to listen for incoming HTTP requests.**

****

Spring Boot Actuator is a powerful module in the Spring Boot ecosystem that provides production-ready features to help monitor and manage your application. It simplifies the process of managing your application by exposing various endpoints that provide valuable insights into its health, metrics, environment, and more.

**Key Features of Spring Boot Actuator**

1. **Health Checks:**
   * **Endpoint:** /actuator/health
   * **Purpose:** Provides information about the health of the application. It performs various checks (e.g., database connectivity, disk space) and returns a status indicating whether the application is up and running properly.
2. **Metrics:**
   * **Endpoint:** /actuator/metrics
   * **Purpose:** Exposes metrics related to the application's performance and health. This includes JVM metrics (e.g., memory usage, garbage collection) and application-specific metrics (e.g., custom counters, gauges).
3. **Application Info:**
   * **Endpoint:** /actuator/info
   * **Purpose:** Displays general information about the application, such as build version, description, and custom metadata. This is often used for displaying application details in monitoring dashboards.
4. **Environment Information:**
   * **Endpoint:** /actuator/env
   * **Purpose:** Provides details about the environment in which the application is running, including system properties and environment variables. Useful for debugging and understanding the application's configuration.
5. **Thread Dump:**
   * **Endpoint:** /actuator/threaddump
   * **Purpose:** Generates and exposes a thread dump of the JVM. This is useful for diagnosing performance issues and understanding thread activity.
6. **Loggers:**
   * **Endpoint:** /actuator/loggers
   * **Purpose:** Allows you to view and modify the logging levels of the application at runtime. This is useful for adjusting logging verbosity without restarting the application.
7. **Auditing Events:**
   * **Endpoint:** /actuator/auditevents
   * **Purpose:** Displays audit events if auditing is enabled in your application. This is useful for tracking user actions or system changes.
8. **Prometheus Metrics:**
   * **Endpoint:** /actuator/prometheus
   * **Purpose:** Provides metrics in a format compatible with Prometheus, allowing you to integrate with Prometheus for advanced monitoring and alerting.

**Configuration**

Actuator endpoints can be configured and enabled or disabled based on your requirements. You can control which endpoints are exposed using properties in application.properties or application.yml.

**Example Configuration:**

**Using application.properties:**

properties

Copy code

# Enable specific Actuator endpoints

management.endpoints.web.exposure.include=health,info,metrics

# Enable all Actuator endpoints

management.endpoints.web.exposure.include=\*

# Enable Actuator over HTTP

management.endpoints.web.enabled=true

**Using application.yml:**

yaml

Copy code

management:

endpoints:

web:

exposure:

include: 'health,info,metrics'

enabled: true

**Security Considerations**

* **Access Control:** By default, not all endpoints are exposed for security reasons. Make sure to secure your Actuator endpoints using Spring Security to prevent unauthorized access. You can use roles or authentication mechanisms to control access.
* **Sensitive Information:** Be cautious with sensitive information exposed by some endpoints. In production environments, limit access to critical endpoints and ensure that only authorized personnel can access them.

**Integration with Monitoring Systems**

Actuator integrates well with monitoring and alerting systems. For instance:

* **Prometheus:** Use the /actuator/prometheus endpoint to expose metrics for Prometheus.
* **Grafana:** Visualize metrics from Prometheus using Grafana dashboards.
* **ELK Stack:** Send logs and metrics to ELK (Elasticsearch, Logstash, Kibana) for advanced logging and analysis.

**Example Use Case**

Imagine you have a Spring Boot application running in production. You want to monitor its health and performance. By enabling the /actuator/health endpoint, you can periodically check if the application is healthy. By using the /actuator/metrics endpoint, you can track metrics like request rates and memory usage. This helps you quickly identify and address issues, ensuring your application runs smoothly.

In summary, Spring Boot Actuator provides essential tools for managing and monitoring Spring Boot applications, enhancing their robustness and maintainability in production environments.

**Spring Boot Auto-Configuration**

Spring Boot's auto-configuration feature simplifies the setup of a Spring application by automatically configuring beans based on the project's classpath, properties, and other conditions. It reduces the need for explicit bean definitions and configuration, making it easier to get started with Spring Boot.

**How Auto-Configuration Works**

1. **Classpath Scanning:**
   * Spring Boot examines the classpath to determine which libraries are present and configures beans based on the detected dependencies. For example, if it detects a dependency for a web server, it will configure web-related beans.
2. **Conditional Annotations:**
   * Auto-configuration is driven by conditional annotations (@ConditionalOnClass, @ConditionalOnMissingBean, etc.) that allow configurations to be applied only when specific conditions are met. This ensures that configurations are only applied if they are relevant to the application.
3. **Configuration Classes:**
   * Auto-configuration classes are typically defined in the META-INF/spring.factories file within the JAR files of Spring Boot starters. These classes are loaded and processed to apply the necessary configurations.

**Example of Auto-Configuration**

Let’s walk through a practical example of how auto-configuration works in a Spring Boot application.

**Scenario: Configuring a DataSource**

Assume you are building a Spring Boot application that requires a database connection. Spring Boot can auto-configure a DataSource bean for you based on the presence of certain dependencies and properties.

1. **Add Dependencies:**

You add the spring-boot-starter-data-jpa starter to your pom.xml file:

xml

Copy code

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

1. **Configuration Properties:**

You define database properties in application.properties:

properties

Copy code

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.hibernate.ddl-auto=update

1. **Auto-Configuration in Action:**
   * **Classpath Scanning:** Spring Boot detects that the spring-boot-starter-data-jpa dependency is on the classpath, which includes libraries for JPA and data source management.
   * **Conditional Configuration:** Spring Boot's auto-configuration for JPA and DataSource is triggered because it finds the necessary classes (DataSource, EntityManagerFactory, etc.) on the classpath and configuration properties in application.properties.
   * **Bean Creation:** Based on these properties, Spring Boot auto-configures a DataSource bean and a EntityManagerFactory bean. It sets up the database connection and JPA entity management without requiring manual bean definitions.

**Auto-Configuration Class Example**

Let’s look at a simplified example of an auto-configuration class for a DataSource:

java

Copy code

@Configuration

@ConditionalOnClass(DataSource.class)

@ConditionalOnMissingBean(DataSource.class)

public class DataSourceAutoConfiguration {

@Bean

@ConfigurationProperties(prefix = "spring.datasource")

public DataSource dataSource() {

return new DriverManagerDataSource();

}

}

* **@Configuration:** Marks this class as a configuration class.
* **@ConditionalOnClass(DataSource.class):** This configuration is only applied if DataSource is on the classpath.
* **@ConditionalOnMissingBean(DataSource.class):** This bean is only created if a DataSource bean does not already exist in the application context.
* **@Bean:** Defines a DataSource bean with properties loaded from application.properties.

**Customizing Auto-Configuration**

While auto-configuration provides sensible defaults, you might want to customize or override some of the auto-configured beans:

1. **Custom Configuration:**
   * Define your own DataSource bean in a configuration class:

java

Copy code

@Configuration

public class CustomDataSourceConfig {

@Bean

public DataSource dataSource() {

// Custom DataSource configuration

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");

dataSource.setUsername("user");

dataSource.setPassword("password");

return dataSource;

}

}

1. **Excluding Auto-Configuration:**
   * You can exclude specific auto-configuration classes using the exclude attribute of @SpringBootApplication:

java

Copy code

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

**Summary**

Spring Boot’s auto-configuration feature automatically sets up your application based on the dependencies and configuration properties provided. This reduces the need for manual configuration and speeds up development, making it easier to create Spring-based applications with minimal setup. However, it also provides flexibility for customization when needed.

**Explain flow how Spring boot works internally in case of a RestAPI**

To understand how Spring Boot works internally when handling a REST API request, it's useful to look at the flow from application startup to handling an incoming HTTP request. Here's a detailed breakdown:

**1. Application Startup**

1. **Bootstrap Class:**
   * The entry point of a Spring Boot application is the main method annotated with @SpringBootApplication. This annotation is a combination of @Configuration, @EnableAutoConfiguration, and @ComponentScan.

java

Copy code

@SpringBootApplication

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

1. **SpringApplication Initialization:**
   * SpringApplication.run() initializes the Spring ApplicationContext. This process includes:
     + **Creating the ApplicationContext:** It sets up the AnnotationConfigApplicationContext or GenericWebApplicationContext.
     + **Loading Configuration:** It loads configuration classes and properties.
     + **Running Initializers:** It runs any application context initializers.
2. **Auto-Configuration:**
   * Spring Boot’s auto-configuration mechanism scans the classpath and configures beans based on the detected dependencies and settings. For example, if spring-boot-starter-web is on the classpath, it configures a DispatcherServlet, RequestMappingHandlerAdapter, and other web-related beans.
3. **Component Scanning:**
   * The @ComponentScan annotation automatically scans the classpath for components (e.g., @RestController, @Service, @Repository) and registers them as beans in the ApplicationContext.

**2. Handling an HTTP Request**

1. **DispatcherServlet Initialization:**
   * The DispatcherServlet is a core component in Spring MVC. It is registered by default in a Spring Boot application when you include the spring-boot-starter-web dependency.
   * The DispatcherServlet acts as the front controller for handling HTTP requests.
2. **Request Mapping:**
   * When an HTTP request is made, the DispatcherServlet receives it and delegates it to the appropriate handler based on the request URL and HTTP method.
   * Spring Boot uses RequestMappingHandlerMapping to map the request to the correct @RequestMapping or @GetMapping annotated method in a @RestController.
3. **Controller Execution:**
   * The request is handled by the appropriate controller method. For instance, if you have a REST controller like this:

java

Copy code

@RestController

public class MyController {

@GetMapping("/api/greet")

public ResponseEntity<String> greet() {

return ResponseEntity.ok("Hello, World!");

}

}

* + The @GetMapping("/api/greet") method handles GET requests to /api/greet.

1. **Handler Execution:**
   * The RequestMappingHandlerAdapter invokes the method on the controller. It also handles any method arguments and response conversion:
     + **Argument Resolvers:** It resolves method parameters (e.g., @RequestParam, @PathVariable, @RequestBody).
     + **Return Value Handlers:** It handles the method return value and converts it to the appropriate response format (e.g., JSON or XML) using HttpMessageConverter.
2. **Response Generation:**
   * The response from the controller method is converted to the desired format and returned to the client.
   * For example, if the controller method returns a String, it will be converted to the response body.
3. **Exception Handling:**
   * If an exception occurs during request processing, Spring Boot provides built-in exception handling. It can return a default error response or use @ControllerAdvice to handle exceptions globally.

**Flow Summary:**

1. **Startup:**
   * SpringApplication.run() initializes the ApplicationContext.
   * Auto-configuration sets up web components, including DispatcherServlet.
   * Component scanning registers controllers and other beans.
2. **Request Handling:**
   * DispatcherServlet receives the HTTP request.
   * RequestMappingHandlerMapping maps the request to a controller method.
   * RequestMappingHandlerAdapter invokes the controller method.
   * The method processes the request and returns a response.
   * Response is converted and sent back to the client.

**Here are the answers to the 50 Spring Boot interview questions:**

**Basic Spring Boot Concepts**

1. **What is Spring Boot, and how does it differ from traditional Spring Framework?**
   * **Answer: Spring Boot is an extension of the Spring Framework that simplifies the setup of new Spring applications. It offers convention over configuration, auto-configuration, and an embedded server, allowing developers to get started quickly without needing extensive configuration or boilerplate code. Unlike traditional Spring, which requires detailed XML configuration or Java-based configuration, Spring Boot auto-configures components based on the dependencies in the project.**
2. **Explain the purpose of @SpringBootApplication. What annotations does it combine?**
   * **Answer: @SpringBootApplication is a convenience annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. It marks the main class of a Spring Boot application and enables auto-configuration, component scanning, and the definition of extra configuration on the application class.**
3. **What are the advantages of using Spring Boot over traditional Spring MVC?**
   * **Answer: Spring Boot offers several advantages:**
     + **Auto-Configuration: Automatically configures necessary components based on dependencies.**
     + **Embedded Servers: Allows running applications without external server configuration.**
     + **Starter Dependencies: Simplifies dependency management.**
     + **Production-Ready Features: Provides health checks, metrics, and externalized configuration out-of-the-box.**
4. **What is auto-configuration in Spring Boot, and how does it work?**
   * **Answer: Auto-configuration attempts to automatically configure your Spring application based on the jar dependencies you have added. It uses @Conditional annotations to check for the presence of certain classes and then provides default configurations that can be overridden by user-defined beans.**
5. **How does Spring Boot simplify dependency management?**
   * **Answer: Spring Boot uses starter dependencies (e.g., spring-boot-starter-web, spring-boot-starter-data-jpa) that aggregate commonly used dependencies into a single package, reducing the need to manually specify and manage individual dependencies in your project.**
6. **Explain the difference between @ComponentScan and @EntityScan.**
   * **Answer: @ComponentScan is used to scan packages for Spring components (e.g., @Component, @Service, @Repository, @Controller) and register them as beans. @EntityScan is specifically used to scan packages for JPA entity classes.**
7. **What is the significance of application.properties or application.yml in a Spring Boot project?**
   * **Answer: application.properties or application.yml files are used to configure application settings. They can define server ports, database configurations, logging levels, and more. Spring Boot automatically loads these files and uses the defined properties to configure the application.**
8. **How can you override default configurations in Spring Boot?**
   * **Answer: You can override default configurations by specifying custom properties in application.properties or application.yml, defining your own beans, or using the @Configuration annotation to provide custom configurations.**
9. **What is Spring Boot’s embedded server? Which servers are supported, and how do you change the default server?**
   * **Answer: Spring Boot applications can run with embedded servers (e.g., Tomcat, Jetty, Undertow), which means you don’t need to deploy your application to an external server. The default server is Tomcat, but you can change it by excluding the default server dependency and adding the dependency for the desired server in your pom.xml or build.gradle.**
10. **Explain the use of profiles in Spring Boot and how to configure them.**
    * **Answer: Spring Boot profiles allow you to create different configurations for different environments (e.g., dev, test, prod). You can define profile-specific properties in application-dev.properties or application-prod.properties files and activate a profile using the spring.profiles.active property.**

**Advanced Spring Boot Concepts**

1. **What is the role of @ConditionalOnProperty and @ConditionalOnMissingBean annotations? Provide examples.**
   * **Answer: @ConditionalOnProperty configures a bean only if a specified property is present and has a specific value. Example:**

**java**

**Copy code**

**@Bean**

**@ConditionalOnProperty(name = "feature.enabled", havingValue = "true")**

**public FeatureService featureService() {**

**return new FeatureService();**

**}**

**@ConditionalOnMissingBean configures a bean only if a specific bean is not already present in the context. Example:**

**java**

**Copy code**

**@Bean**

**@ConditionalOnMissingBean**

**public MyService myService() {**

**return new MyServiceImpl();**

**}**

1. **Explain how Spring Boot handles externalized configuration using @ConfigurationProperties.**
   * **Answer: @ConfigurationProperties binds the external configuration properties defined in application.properties or application.yml to a POJO. This allows you to group related configurations together and inject them into beans. Example:**

**java**

**Copy code**

**@ConfigurationProperties(prefix = "app.datasource")**

**public class DataSourceProperties {**

**private String url;**

**private String username;**

**private String password;**

**// getters and setters**

**}**

1. **What is the purpose of @EnableAutoConfiguration, and how does it affect the application?**
   * **Answer: @EnableAutoConfiguration enables Spring Boot's auto-configuration feature, which attempts to automatically configure your Spring application based on the jar dependencies you have added. It reduces the need for manual configuration and allows for quick setup of Spring applications.**
2. **How do you configure security in a Spring Boot application? Explain the role of @EnableWebSecurity.**
   * **Answer: Security in Spring Boot can be configured by extending WebSecurityConfigurerAdapter and overriding the configure methods. @EnableWebSecurity activates Spring Security's web security support and provides the integration with Spring MVC. Example:**

**java**

**Copy code**

**@EnableWebSecurity**

**public class SecurityConfig extends WebSecurityConfigurerAdapter {**

**@Override**

**protected void configure(HttpSecurity http) throws Exception {**

**http.authorizeRequests()**

**.antMatchers("/public").permitAll()**

**.anyRequest().authenticated()**

**.and().formLogin();**

**}**

**}**

1. **Describe the different scopes of beans in Spring Boot and how to define them.**
   * **Answer: The common bean scopes in Spring Boot are:**
     + **Singleton: A single instance per Spring IoC container (default scope).**
     + **Prototype: A new instance is created every time it is requested.**
     + **Request: A single instance per HTTP request.**
     + **Session: A single instance per HTTP session.**
     + **Application: A single instance per ServletContext. Scopes can be defined using the @Scope annotation. Example:**

**java**

**Copy code**

**@Bean**

**@Scope("prototype")**

**public MyBean myBean() {**

**return new MyBean();**

**}**

1. **What are Spring Boot starters, and how do they work?**
   * **Answer: Spring Boot starters are dependency descriptors that aggregate multiple dependencies into a single dependency. They simplify dependency management by providing commonly used dependencies for specific functionalities (e.g., spring-boot-starter-web, spring-boot-starter-data-jpa). When you include a starter, it automatically pulls in the necessary dependencies for you.**
2. **How does Spring Boot handle exception management in RESTful services?**
   * **Answer: Spring Boot provides a default exception handling mechanism using @ControllerAdvice and @ExceptionHandler to handle exceptions globally or per controller. It can return custom error responses to clients in a consistent format.**

**java**

**Copy code**

**@ControllerAdvice**

**public class GlobalExceptionHandler {**

**@ExceptionHandler(ResourceNotFoundException.class)**

**public ResponseEntity<ErrorResponse> handleResourceNotFound(ResourceNotFoundException ex) {**

**ErrorResponse response = new ErrorResponse("NOT\_FOUND", ex.getMessage());**

**return new ResponseEntity<>(response, HttpStatus.NOT\_FOUND);**

**}**

**}**

1. **What is Spring Boot Actuator, and what are some of the important metrics and endpoints it provides?**
   * **Answer: Spring Boot Actuator provides production-ready features like monitoring, metrics, and health checks. It exposes various endpoints such as:**
     + **/actuator/health: Shows application health status.**
     + **/actuator/metrics: Provides various application metrics.**
     + **/actuator/env: Displays environment properties.**
     + **/actuator/loggers: Manages and views application logging levels.**
2. **How would you create custom Actuator endpoints?**
   * **Answer: You can create custom Actuator endpoints by implementing the Endpoint interface or extending the AbstractEndpoint class. Example:**

**java**

**Copy code**

**@Component**

**@Endpoint(id = "custom")**

**public class CustomEndpoint {**

**@ReadOperation**

**public CustomData customOperation() {**

**return new CustomData("value1", "value2");**

**}**

**}**

1. **How do you configure and use the Spring Boot DevTools?**
   * **Answer: Spring Boot DevTools provides developer-friendly features like automatic restarts, live reload, and configurations optimized for development. You can include it as a dependency in your project, and it will automatically enable hot swapping, disable caching, and enhance development productivity.**

**Microservices and Spring Boot**

1. **How does Spring Boot support microservices architecture?**
   * **Answer: Spring Boot supports microservices by providing features like RESTful APIs, embedded servers, easy deployment, and integration with Spring Cloud for service discovery, circuit breakers, centralized configuration, and distributed tracing.**
2. **Explain the use of Spring Cloud with Spring Boot in a microservices environment.**
   * **Answer: Spring Cloud extends Spring Boot's capabilities for building microservices. It provides tools for configuration management (Spring Cloud Config), service discovery (Eureka), load balancing (Ribbon), circuit breakers (Hystrix), and API Gateway (Zuul or Spring Cloud Gateway).**
3. **What is a circuit breaker in microservices, and how do you implement it using Spring Boot?**
   * **Answer: A circuit breaker is a pattern used to prevent failure cascading in a microservices architecture by breaking the connection to a failing service temporarily. You can implement it using Spring Boot with Hystrix or Resilience4j. Example using Resilience4j:**

**java**

**Copy code**

**@Service**

**public class MyService {**

**@CircuitBreaker(name = "myService", fallbackMethod = "fallback")**

**public String callRemoteService() {**

**// call remote service**

**}**

**public String fallback(Throwable t) {**

**return "Fallback response";**

**}**

**}**

1. **Describe the role of FeignClient in microservices communication.**
   * **Answer: FeignClient is a declarative HTTP client that simplifies communication between microservices. It integrates with Ribbon for client-side load balancing and with Hystrix for circuit breaking. Example:**

**java**

**Copy code**

**@FeignClient(name = "user-service")**

**public interface UserServiceClient {**

**@GetMapping("/users/{id}")**

**User getUserById(@PathVariable("id") Long id);**

**}**

1. **How does Spring Boot manage distributed configuration with Spring Cloud Config Server?**
   * **Answer: Spring Cloud Config Server provides a centralized configuration for distributed systems. It allows you to manage configurations in a version-controlled repository (e.g., Git) and automatically applies the configurations to all services that use it.**
2. **Explain service discovery in Spring Boot using Eureka.**
   * **Answer: Eureka is a service registry provided by Netflix OSS and integrated into Spring Cloud. It allows microservices to register themselves at runtime as they come up and also allows other services to discover them using the service registry.**
3. **What is the role of API Gateway in a microservices architecture, and how can you implement it using Spring Boot?**
   * **Answer: An API Gateway acts as a single entry point for all client requests, handling routing, load balancing, and sometimes security and rate limiting. You can implement an API Gateway using Spring Cloud Gateway or Zuul in a Spring Boot application.**
4. **Discuss the importance of tracing and monitoring in microservices. How does Spring Boot help in this regard?**
   * **Answer: Tracing and monitoring are crucial in microservices to understand the flow of requests, identify bottlenecks, and debug issues. Spring Boot integrates with distributed tracing tools like Zipkin and Sleuth to provide request tracking across microservices.**
5. **How do you handle transaction management in a microservices architecture using Spring Boot?**
   * **Answer: In a microservices architecture, distributed transactions are managed using patterns like Saga or 2-phase commit. Spring Boot can help by implementing compensating transactions or leveraging tools like Spring Cloud Data Flow for orchestrating long-running processes.**
6. **How would you design a resilient microservices architecture using Spring Boot?**
   * **Answer: Designing a resilient microservices architecture involves using patterns like circuit breakers, retries, timeouts, bulkheads, and fallback mechanisms. Spring Boot, along with Spring Cloud, provides tools like Resilience4j, Hystrix, and load balancing to implement these patterns.**

**Data Access and Persistence**

1. **How does Spring Boot simplify working with databases? Explain the use of Spring Data JPA.**
   * **Answer: Spring Boot simplifies database access through auto-configuration of DataSource, JPA, and ORM frameworks. Spring Data JPA provides an abstraction over JPA, allowing you to create repositories with minimal boilerplate code, using method naming conventions to generate SQL queries automatically.**
2. **What is the role of @Entity, @Table, and @Repository annotations in Spring Boot?**
   * **Answer:**
     + **@Entity: Marks a class as a JPA entity to be mapped to a database table.**
     + **@Table: Specifies the table name for the entity if it differs from the class name.**
     + **@Repository: Marks a class as a Data Access Object (DAO) and provides exception translation from SQLExceptions to Spring's DataAccessException.**
3. **Explain the concept of lazy loading and eager loading in JPA. How do you configure it in Spring Boot?**
   * **Answer:**
     + **Lazy Loading: Fetches related entities on demand (when accessed for the first time).**
     + **Eager Loading: Fetches related entities immediately with the main entity. You configure it using FetchType.LAZY or FetchType.EAGER in associations (@OneToMany, @ManyToOne, etc.). Example:**

**java**

**Copy code**

**@ManyToOne(fetch = FetchType.LAZY)**

**private Department department;**

1. **How can you optimize database queries using Spring Boot and JPA?**
   * **Answer: Optimization techniques include:**
     + **Using @Query for custom and optimized queries.**
     + **Implementing pagination and limiting result sets.**
     + **Using @NamedEntityGraph for controlling entity loading.**
     + **Indexing database columns used in queries.**
     + **Caching frequently accessed data using Spring Cache.**
2. **Describe how to handle transactions in Spring Boot.**
   * **Answer: Transactions in Spring Boot are managed using @Transactional. It can be applied at the class or method level to demarcate transactional boundaries. Spring Boot supports declarative transaction management, and you can configure transaction propagation and isolation levels using attributes of @Transactional.**
3. **What is the difference between CrudRepository, JpaRepository, and PagingAndSortingRepository?**
   * **Answer:**
     + **CrudRepository: Provides CRUD operations.**
     + **PagingAndSortingRepository: Extends CrudRepository to provide additional methods for pagination and sorting.**
     + **JpaRepository: Extends PagingAndSortingRepository to include JPA-specific operations like flushing the persistence context and batch processing.**
4. **How do you implement pagination and sorting in Spring Boot?**
   * **Answer: You can implement pagination and sorting by extending PagingAndSortingRepository or JpaRepository and using the Pageable and Sort objects in method parameters. Example:**

**java**

**Copy code**

**Page<User> findByLastName(String lastName, Pageable pageable);**

**List<User> findByLastName(String lastName, Sort sort);**

1. **What are the best practices for managing database migrations in Spring Boot applications?**
   * **Answer: Use tools like Flyway or Liquibase for version-controlled database migrations. Define migration scripts in /db/migration directory, and Spring Boot will automatically execute them on application startup.**
2. **Explain how to use NoSQL databases like MongoDB with Spring Boot.**
   * **Answer: Spring Boot provides support for NoSQL databases like MongoDB through Spring Data MongoDB. You can define MongoDB repositories by extending MongoRepository, configure the connection in application.properties, and map your documents using annotations like @Document and @Field.**
3. **How can you create and manage custom queries in Spring Data JPA?**
   * **Answer: You can create custom queries using the @Query annotation, where you can define JPQL or native SQL queries. Example:**

**java**

**Copy code**

**@Query("SELECT u FROM User u WHERE u.email = ?1")**

**User findByEmail(String email);**

**Testing and Deployment**

1. **What are the different types of tests you can write in Spring Boot? Explain the role of @SpringBootTest.**
   * **Answer: Spring Boot supports various types of tests:**
     + **Unit Tests: Testing individual components with mock dependencies.**
     + **Integration Tests: Testing the application with a real application context.**
     + **End-to-End Tests: Testing the complete application flow. @SpringBootTest loads the full application context, allowing integration tests to run in a Spring environment.**
2. **How do you mock dependencies in Spring Boot tests? Explain the use of @MockBean and @SpyBean.**
   * **Answer: @MockBean creates and injects a mock instance of a bean into the application context for testing purposes, replacing the actual bean. @SpyBean creates and injects a spy of a bean, allowing you to override specific method behaviors while keeping the original behavior for other methods.**
3. **What is the purpose of @TestConfiguration, and how does it differ from @Configuration?**
   * **Answer: @TestConfiguration is a specialized form of @Configuration that is used to define beans and configuration specifically for test cases. It is only loaded during testing, unlike @Configuration, which is used for application-wide configurations.**
4. **Explain how to use Testcontainers for integration testing in Spring Boot.**
   * **Answer: Testcontainers is a Java library that provides lightweight, throwaway instances of common databases, message brokers, and other services running in Docker containers for integration testing. It allows you to spin up a real database or service in a Docker container and run your tests against it. Example:**

**java**

**Copy code**

**@Testcontainers**

**public class MyIntegrationTest {**

**@Container**

**public static PostgreSQLContainer<?> postgresqlContainer = new PostgreSQLContainer<>("postgres:latest");**

**}**

1. **Describe how to deploy a Spring Boot application to a cloud platform like AWS or Azure.**
   * **Answer: Deploying to a cloud platform involves packaging your Spring Boot application as a jar or war file and deploying it on cloud services like AWS Elastic Beanstalk, Azure App Service, or Kubernetes. You can use CI/CD pipelines with Jenkins or GitHub Actions for automated deployments.**
2. **What are the best practices for securing a Spring Boot application before deployment?**
   * **Answer: Best practices include:**
     + **Securing endpoints with Spring Security.**
     + **Encrypting sensitive properties using Jasypt or Spring Cloud Vault.**
     + **Validating and sanitizing inputs to prevent SQL injection and XSS.**
     + **Using HTTPS and configuring CORS.**
     + **Regularly updating dependencies and applying security patches.**
3. **How do you externalize configuration for different environments (e.g., dev, test, prod) in Spring Boot?**
   * **Answer: Use profile-specific properties files like application-dev.properties or application-prod.properties. You can also use environment variables or command-line arguments to override properties for different environments.**
4. **What is the purpose of spring-boot-maven-plugin or spring-boot-gradle-plugin?**
   * **Answer: These plugins provide build support for Spring Boot applications. They package the application as an executable jar or war, run the application, and manage dependencies.**
5. **How would you implement logging in a Spring Boot application?**
   * **Answer: Logging in Spring Boot can be implemented using SLF4J with Logback as the default logging framework. Configure logging levels and patterns in application.properties, and use @Slf4j to inject a logger instance into your classes.**
6. **How do you handle application versioning and backward compatibility in Spring Boot?**
   * **Answer: Handle versioning by using semantic versioning (e.g., 1.0.0), maintaining backward compatibility by carefully deprecating old features and introducing new ones without breaking existing functionality. You can use Spring Cloud Contract for contract testing between services to ensure compatibility.**